Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Unlike message queues and sockets, shared memory involves no kernel copying operations: ***its fast!***

Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Unlike message queues and sockets, shared memory involves no kernel copying operations: ***its fast!***
- Multiple processes access the same memory; to prevent race conditions, partial memory writes/reads, *use semaphores*

Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Unlike message queues and sockets, shared memory involves no kernel copying operations: ***its fast!***
- Multiple processes access the same memory; to prevent race conditions, partial memory writes/reads, *use semaphores*
- There are two forms of shared memory:

Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Unlike message queues and sockets, shared memory involves no kernel copying operations: ***its fast!***
- Multiple processes access the same memory; to prevent race conditions, partial memory writes/reads, *use semaphores*
- There are two forms of shared memory:
  1. all-in-ram (which will be considered first as “shared memory”)

Shared Memory

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Unlike message queues and sockets, shared memory involves no kernel copying operations: ***its fast!***
- Multiple processes access the same memory; to prevent race conditions, partial memory writes/reads, *use semaphores*
- There are two forms of shared memory:
  1. all-in-ram (which will be considered first as “shared memory”)
  2. memory mapped to a file

Shared Memory Issues

When one has many CPUs *(cores)*...

- Memory generally is accessed over a common bus, or from an on-chip cache, and only permits one access at a time.

Shared Memory Issues

When one has many CPUs *(cores)*...

- Memory generally is accessed over a common bus, or from an on-chip cache, and only permits one access at a time.
- Consequently, multiple CPUs accessing the same memory can encounter a bottleneck.

Shared Memory Issues

When one has many CPUs *(cores)*...

- Memory generally is accessed over a common bus, or from an on-chip cache, and only permits one access at a time.
- Consequently, multiple CPUs accessing the same memory can encounter a bottleneck.
- When multiple copies of memory are available, care must be taken to insure that the memory (and cache) must be kept coherent *(ie. changes made by a CPU to the information needs to be reflected to the other CPUs)*

Shared Memory Issues

When one has many CPUs *(cores)*...

- Memory generally is accessed over a common bus, or from an on-chip cache, and only permits one access at a time.
- Consequently, multiple CPUs accessing the same memory can encounter a bottleneck.
- When multiple copies of memory are available, care must be taken to insure that the memory (and cache) must be kept coherent *(ie. changes made by a CPU to the information needs to be reflected to the other CPUs)*
- Consequently, in multiple CPU environments, sometimes messaging provides improved performance, in spite of the extra copying of information that is involved.

Shared Memory Issues

When one has many CPUs *(cores)*...

- Memory generally is accessed over a common bus, or from an on-chip cache, and only permits one access at a time.
- Consequently, multiple CPUs accessing the same memory can encounter a bottleneck.
- When multiple copies of memory are available, care must be taken to insure that the memory (and cache) must be kept coherent *(ie. changes made by a CPU to the information needs to be reflected to the other CPUs)*
- Consequently, in multiple CPU environments, sometimes messaging provides improved performance, in spite of the extra copying of information that is involved.
- These issues are magnified in fine-grain threading.

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory
- shmflg may be an OR'd combination of:

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory
- shmflg may be an OR'd combination of:

SHM_R	readable by owner
SHM_W	writable by owner

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory
- shmflg may be an OR'd combination of:

SHM_R	readable by owner
SHM_W	writable by owner
SHM_R>> 3	readable by group
SHM_W>> 3	writable by group

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory
- shmflg may be an OR'd combination of:

SHM_R	readable by owner
SHM_W	writable by owner
SHM_R>> 3	readable by group
SHM_W>> 3	writable by group
SHM_R>> 6	readable by others
SHM_W>> 6	writable by others

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory
- shmflg may be an OR'd combination of:

SHM_R	readable by owner
SHM_W	writable by owner
SHM_R>> 3	readable by group
SHM_W>> 3	writable by group
SHM_R>> 6	readable by others
SHM_W>> 6	writable by others
IPC_CREAT	these work as they
IPC_EXCL	do for MQs and semaphores

Shared Memory: shmget()

int shmget(key_t key, size_t size, int shmflg);

- Returns shmid, an identifier associated with the shared memory
- shmflg may be an OR'd combination of:

SHM_R	readable by owner
SHM_W	writable by owner
SHM_R>> 3	readable by group
SHM_W>> 3	writable by group
SHM_R>> 6	readable by others
SHM_W>> 6	writable by others
IPC_CREAT	these work as they
IPC_EXCL	do for MQs and semaphores

- This call creates a shared memory pool, but *does not give your process access* to that pool of memory. The shared memory has not yet been attached to your process' (virtual) memory space.

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():
- shmat() attaches the shared memory segment identified by shmid to the address space of the calling process.

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():
- shmat() attaches the shared memory segment identified by shmid to the address space of the calling process.
- If shmaddr is NULL: the system will choose a suitable address at which to attach the segment. Note that this rules out the use of pointers in the shared memory itself, as different processes will typically attach the segment in different locations.

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():
- shmat() attaches the shared memory segment identified by shmid to the address space of the calling process.
- If shmaddr is NULL: the system will choose a suitable address at which to attach the segment. Note that this rules out the use of pointers in the shared memory itself, as different processes will typically attach the segment in different locations.
- If shmaddr isn't NULL, and shmflg has SHM_RND, then the attachment occurs at the specified address rounded down to the nearest multiple of SHMLBA. *(this is defined as PAGE_SIZE on my Scientific Linux system, which in turn is defined as 0x400)*

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():
- shmat() attaches the shared memory segment identified by shmid to the address space of the calling process.
- If shmaddr is NULL: the system will choose a suitable address at which to attach the segment. Note that this rules out the use of pointers in the shared memory itself, as different processes will typically attach the segment in different locations.
- If shmaddr isn't NULL, and shmflg has SHM_RND, then the attachment occurs at the specified address rounded down to the nearest multiple of SHMLBA. *(this is defined as PAGE_SIZE on my Scientific Linux system, which in turn is defined as 0x400)*
- Otherwise, shmaddr must be a page-aligned address *(getconf PAGESIZE)*

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():
- shmat() attaches the shared memory segment identified by shmid to the address space of the calling process.
- If shmaddr is NULL: the system will choose a suitable address at which to attach the segment. Note that this rules out the use of pointers in the shared memory itself, as different processes will typically attach the segment in different locations.
- If shmaddr isn't NULL, and shmflg has SHM_RND, then the attachment occurs at the specified address rounded down to the nearest multiple of SHMLBA. *(this is defined as PAGE_SIZE on my Scientific Linux system, which in turn is defined as 0x400)*
- Otherwise, shmaddr must be a page-aligned address *(getconf PAGESIZE)*
- If SHM_RDONLY is specified in shmflg, the process must have read permission for the segment; otherwise, the segment is assumed to need both read and write permissions.

Shared Memory: shmat()

void *shmat(int shmid, const void *shmaddr, int shmflg);

- To give your process access to the shared memory pool, use shmat():
- shmat() attaches the shared memory segment identified by shmid to the address space of the calling process.
- If shmaddr is NULL: the system will choose a suitable address at which to attach the segment. Note that this rules out the use of pointers in the shared memory itself, as different processes will typically attach the segment in different locations.
- If shmaddr isn't NULL, and shmflg has SHM_RND, then the attachment occurs at the specified address rounded down to the nearest multiple of SHMLBA. *(this is defined as PAGE_SIZE on my Scientific Linux system, which in turn is defined as 0x400)*
- Otherwise, shmaddr must be a page-aligned address *(getconf PAGESIZE)*
- If SHM_RDONLY is specified in shmflg, the process must have read permission for the segment; otherwise, the segment is assumed to need both read and write permissions.
- After fork(), the child inherits the parent's attached shared memory segments.

Shared Memory: sizing

- For picking the amount of shared memory, I suggest specifying a structure which encompasses everything you want in shared memory:

Shared Memory: sizing

- For picking the amount of shared memory, I suggest specifying a structure which encompasses everything you want in shared memory:

```
struct SharedMemoryPool_str {
    struct ...;
    struct ...;
    ...
    }*shmpool;
```

Shared Memory: sizing

- For picking the amount of shared memory, I suggest specifying a structure which encompasses everything you want in shared memory:

```
struct SharedMemoryPool_str {
        struct ...;
        struct ...;
        ...
        }*shmpool;
shmid= shmget(key,sizeof(struct SharedMemoryPool_str),IPC_CREAT);
```

Shared Memory: sizing

- For picking the amount of shared memory, I suggest specifying a structure which encompasses everything you want in shared memory:

```
struct SharedMemoryPool_str {
        struct ...;
        struct ...;
        ...
        }*shmpool;
shmid= shmget(key,sizeof(struct SharedMemoryPool_str),IPC_CREAT);
shmpool= (struct SharedMemoryPool_str *) shmat(shmid,NULL,0);
```

Shared Memory: sizing

- For picking the amount of shared memory, I suggest specifying a structure which encompasses everything you want in shared memory:

```
struct SharedMemoryPool_str {
        struct ...;
        struct ...;
        ...
        }*shmpool;
shmid= shmget(key,sizeof(struct SharedMemoryPool_str),IPC_CREAT);
shmpool= (struct SharedMemoryPool_str *) shmat(shmid,NULL,0);
```

- Alternative: Silicon Graphics came up with an allocatable shared memory scheme. I have emulated some of it; see http://www.drchip.org/astronaut/src/index.html#USARENA. A subset of the functions available:

Shared Memory: sizing

- For picking the amount of shared memory, I suggest specifying a structure which encompasses everything you want in shared memory:

```
struct SharedMemoryPool_str {
        struct ...;
        struct ...;
        ...
        }*shmpool;
shmid= shmget(key,sizeof(struct SharedMemoryPool_str),IPC_CREAT);
shmpool= (struct SharedMemoryPool_str *) shmat(shmid,NULL,0);
```

- Alternative: Silicon Graphics came up with an allocatable shared memory scheme. I have emulated some of it; see http://www.drchip.org/astronaut/src/index.html#USARENA. A subset of the functions available:

Configuration	ptrdiff_t usconfig(int,...)
Initialization	usptr_t *usinit(const char *)
Allocation	void *uscalloc(size_t, size_t, usptr_t *)
	void usfree(void *, usptr_t *)
	void *usmalloc(size_t, usptr_t *)
	void *usrealloc(void *, size_t, usptr_t *)
	void *usrecalloc(void *, size_t, size_t, usptr_t *)

Shared Memory: Using It

- This is easy! Using our example:

```
shmpool->str1.whatever= whatever; shmpool->str2.whatever= whatever;
```

Shared Memory: Using It

- This is easy! Using our example:

```
shmpool->str1.whatever= whatever; shmpool->str2.whatever= whatever;
```

- However, bear in mind that you really should use semaphores to control access

Shared Memory: Using It

- This is easy! Using our example:

```
shmpool->str1.whatever= whatever; shmpool->str2.whatever= whatever;
```

- However, bear in mind that you really should use semaphores to control access
- So, with semaphores:

```
sops.sem_num= 0;                    // pick a semaphore
sops.sem_op= -1;                    // attempt to decrement the semaphore
sops.sem_flg= 0;                    // block if semaphore can't be decremented
semop(semid,&sops,(size_t) 1);      // "obtain" the semaphore
  shmpool->str1.whatever= whatever; // access the shared memory
  sops.sem_op= 1;                   // increment the semaphore
  semop(semid,&sops,(size_t) 1);    // release the semaphore
```

Shared Memory: deleting it

int shmdt(const void *shmaddr);

- To detach a shared memory segment from your process, use:

```
shmdt(shmpool);
```

(shmdt detaches/unmaps the shared memory from the process' virtual memory pool)

Shared Memory: deleting it

int shmdt(const void *shmaddr);

- To detach a shared memory segment from your process, use:

  shmdt(shmpool);

  (shmdt detaches/unmaps the shared memory from the process' virtual memory pool)

- To remove the shared memory from the system, use:

  shmctl(shmid,IPC_RMID,NULL);

  note that this command will not remove shared memory that still has attachments to it

Shared Memory: shmctl()

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

IPC_STAT copy status of shared memory into buf

Shared Memory: shmctl()

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

IPC_STAT	copy status of shared memory into buf
IPC_SET	Use buf's entries to change the kernel's status information on the shared memory

Shared Memory: shmctl()

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

IPC_STAT	copy status of shared memory into buf
IPC_SET	Use buf's entries to change the kernel's status information on the shared memory
IPC_RMID	mark the segment for destruction (ie. remove the shared memory from the system)
	The segment won't actually be destroyed until the last process detaches it.
	The caller must must be the owner/creator/privileged.

Shared Memory: shmctl()

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

IPC_STAT	copy status of shared memory into buf
IPC_SET	Use buf's entries to change the kernel's status information on the shared memory
IPC_RMID	mark the segment for destruction (ie. remove the shared memory from the system)
	The segment won't actually be destroyed until the last process detaches it.
	The caller must must be the owner/creator/privileged.

```
struct shmid_ds {
        struct ipc_perm shm_perm;    Ownership and permissions
        size_t shm_segsz;            Size of segment (bytes)
        time_t shm_atime;            Last attach time
        time_t shm_dtime;            Last detach time
        time_t shm_ctime;            Last change time
        pid_t shm_cpid;              PID of creator
        pid_t shm_lpid;              PID of last shmat/shmdt
        shmatt_t shm_nattch;         Qty of current attaches
          ...                          ...
    }
```

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

addr specify starting address of the buffer to be mapped.

if null, the kernel will choose the address at which to create the mapping.

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

addr specify starting address of the buffer to be mapped.

if null, the kernel will choose the address at which to create the mapping.

length quantity of bytes to be mapped

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

addr specify starting address of the buffer to be mapped.

if null, the kernel will choose the address at which to create the mapping.

length quantity of bytes to be mapped

prot (see next slide)

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

addr specify starting address of the buffer to be mapped.

if null, the kernel will choose the address at which to create the mapping.

length quantity of bytes to be mapped

prot (see next slide)

flags (see next slide)

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

addr specify starting address of the buffer to be mapped.
if null, the kernel will choose the address at which to create the mapping.

length quantity of bytes to be mapped

prot (see next slide)

flags (see next slide)

fd file descriptor of file to be mapped *(must be open prior to calling mmap)*

Memory Mapped I/O : mmap()

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags,int fd, off_t offset);
```

- Lets us map a file on disk onto a buffer in memory
- Reading from the buffer actually reads from the associated file on the disk
- Writing to the buffer actually writes to the associated file on the disk

addr specify starting address of the buffer to be mapped.
if null, the kernel will choose the address at which to create the mapping.

length quantity of bytes to be mapped

prot (see next slide)

flags (see next slide)

fd file descriptor of file to be mapped *(must be open prior to calling mmap)*

offset starting offset in file for the mapping region *(often zero)*
Must be a multiple of the page size as returned by
sysconf(_SC_PAGESIZE)

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE no access (Sys V)

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE	no access (Sys V)
PROT_READ	region may be read

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE	no access (Sys V)
PROT_READ	region may be read
PROT_WRITE	region may be written

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE	no access (Sys V)
PROT_READ	region may be read
PROT_WRITE	region may be written
PROT_EXEC	region may be executed

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE	no access (Sys V)
PROT_READ	region may be read
PROT_WRITE	region may be written
PROT_EXEC	region may be executed

flags determines whether updates are visible to other processes, and whether updates are passed through to the file.

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE	no access (Sys V)
PROT_READ	region may be read
PROT_WRITE	region may be written
PROT_EXEC	region may be executed

flags determines whether updates are visible to other processes, and whether updates are passed through to the file.

MAP_SHARED	share this mapping. Updates visible to other processes, and the underlying file is updated

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

PROT_NONE	no access (Sys V)
PROT_READ	region may be read
PROT_WRITE	region may be written
PROT_EXEC	region may be executed

flags determines whether updates are visible to other processes, and whether updates are passed through to the file.

MAP_SHARED	share this mapping. Updates visible to other processes, and the underlying file is updated
MAP_PRIVATE	create a private copy-on-write mapping. Updates are not visible to other processes. Changes don't carry through to the underlying file

Memory Mapped I/O, con't.

prot protection spec: describes memory protection, must not conflict with open mode of file. *(you may bitwise-or these together)*

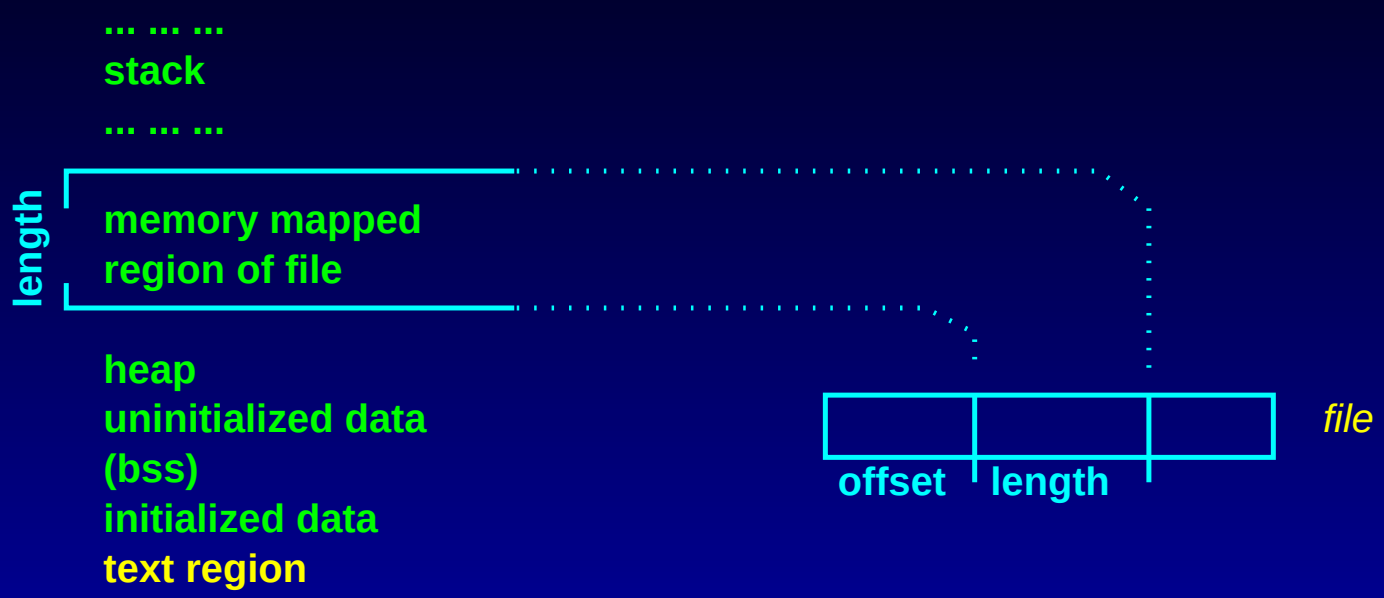
PROT_NONE	no access (Sys V)
PROT_READ	region may be read
PROT_WRITE	region may be written
PROT_EXEC	region may be executed

flags determines whether updates are visible to other processes, and whether updates are passed through to the file.

MAP_SHARED	share this mapping. Updates visible to other processes, and the underlying file is updated
MAP_PRIVATE	create a private copy-on-write mapping. Updates are not visible to other processes. Changes don't carry through to the underlying file

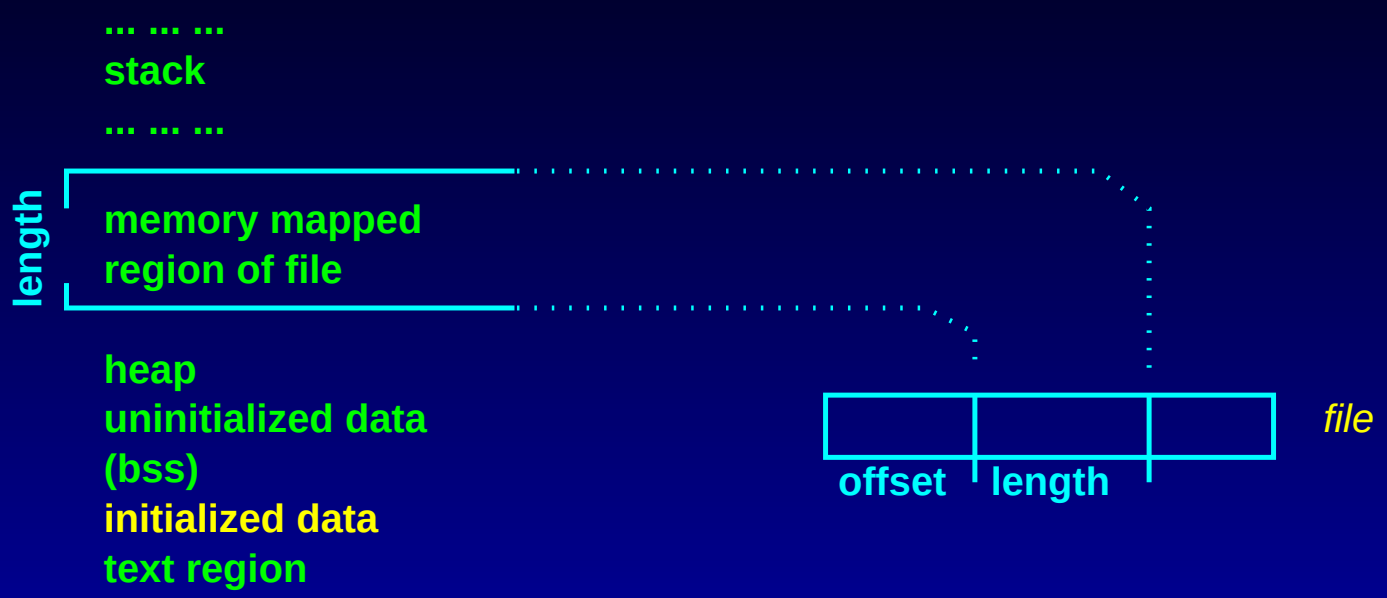
Changes may not actually be updated to the file until msync() or munmap() is called.

Memory Mapped I/O: a picture



text region = code segment holds binary executable code

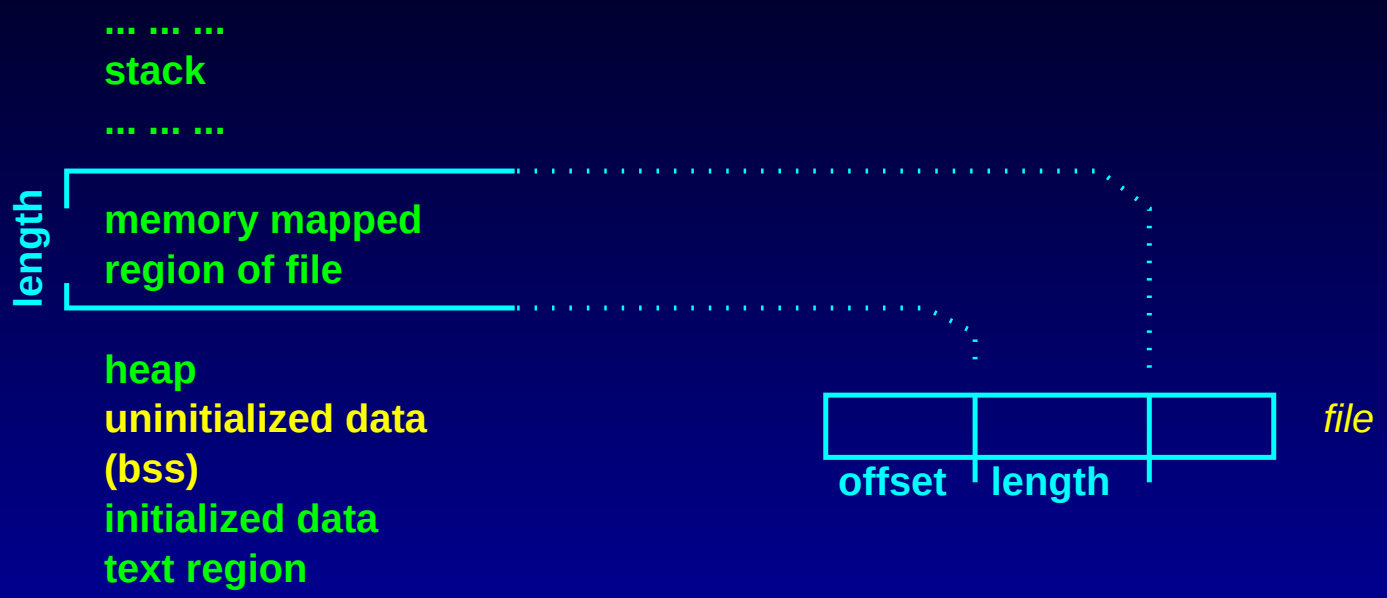
Memory Mapped I/O: a picture



text region = code segment holds binary executable code

data segment holds initialized data

Memory Mapped I/O: a picture

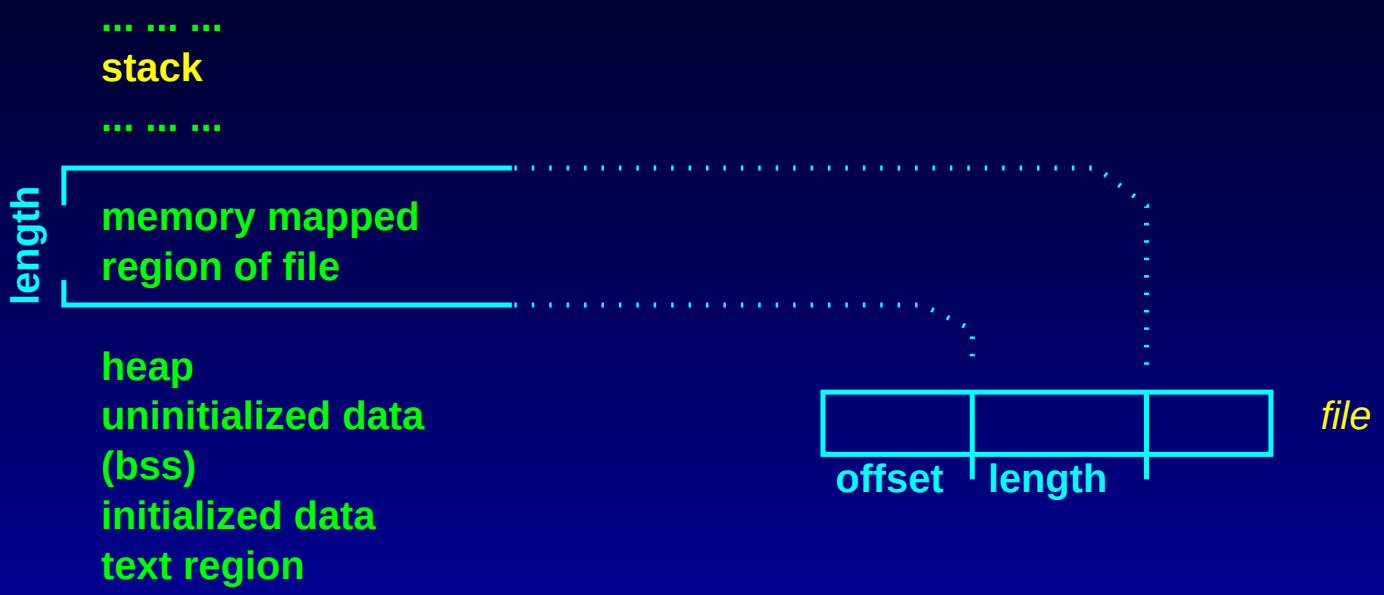


text region = code segment holds binary executable code

data segment holds initialized data

bss block started by symbol (*uninitialized data*)

Memory Mapped I/O: a picture



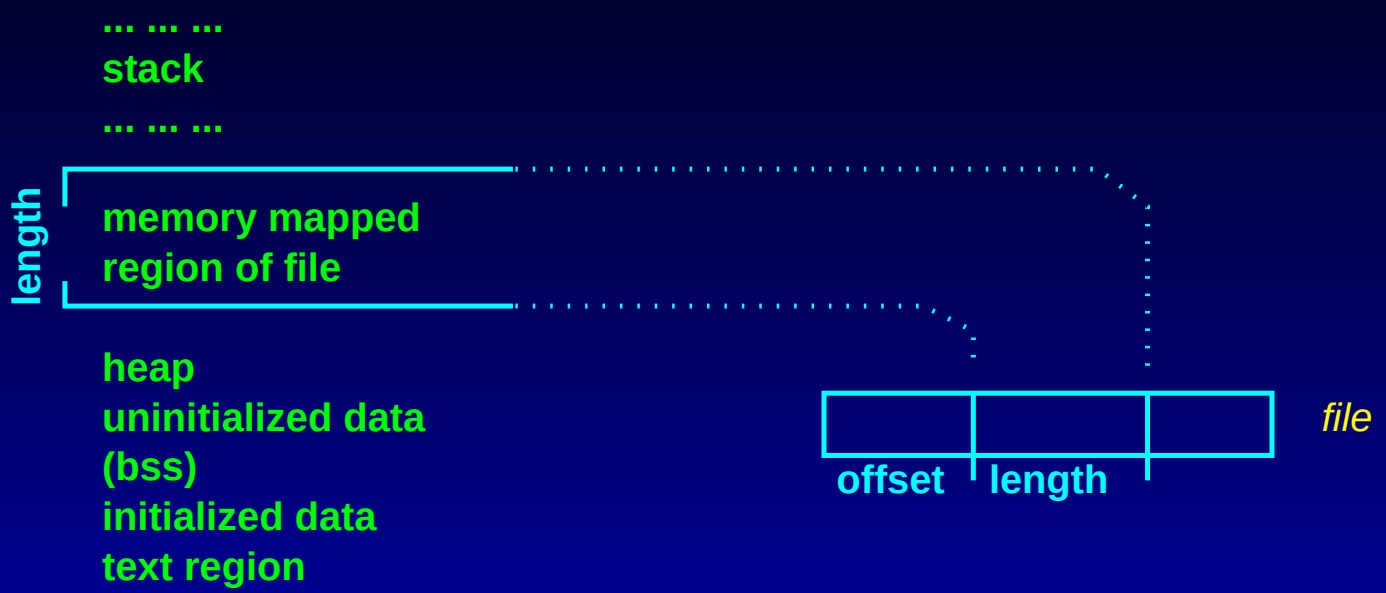
text region = code segment holds binary executable code

data segment holds initialized data

bss block started by symbol (*uninitialized data*)

stack segment holds the stack

Memory Mapped I/O: a picture



text region = code segment holds binary executable code

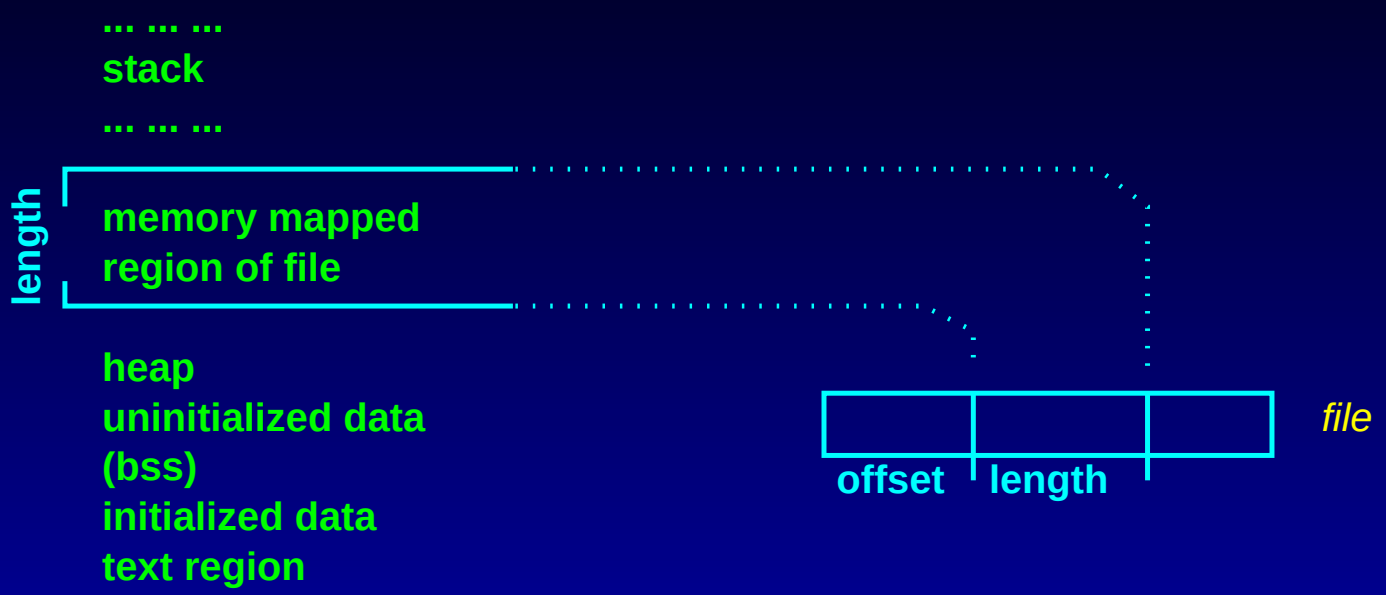
data segment holds initialized data

bss block started by symbol (*uninitialized data*)

stack segment holds the stack

- mmap() does not do memory allocation

Memory Mapped I/O: a picture



text region = code segment holds binary executable code

data segment holds initialized data

bss block started by symbol (*uninitialized data*)

stack segment holds the stack

- mmap() does not do memory allocation
- Newly opened files should be written to at the offset+length-1 byte prior to use (else you'll get a SIGBUS)

Memory Mapped I/O: munmap()

```
int munmap(void *addr, size_t length);
```

- Disassociates the mapped region from the process' virtual address space.

Memory Mapped I/O: munmap()

int munmap(void *addr, size_t length);

- Disassociates the mapped region from the process' virtual address space.
- May be used explicitly or implicitly via program termination

Memory Mapped I/O: munmap()

```
int munmap(void *addr, size_t length);
```

- Disassociates the mapped region from the process' virtual address space.
- May be used explicitly or implicitly via program termination
- Subsequent references to addresses in the mapped region will generate invalid memory reference signals.

Memory Mapped I/O: munmap()

```
int munmap(void *addr, size_t length);
```

- Disassociates the mapped region from the process' virtual address space.
- May be used explicitly or implicitly via program termination
- Subsequent references to addresses in the mapped region will generate invalid memory reference signals.
- Note: closing the file descriptor does not unmap the region!

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.
- length=0: all pages containing addr are synchronized.

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.
- length=0: all pages containing addr are synchronized.
- Otherwise, addr to addr+length bytes are synchronized.

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.
- length=0: all pages containing addr are synchronized.
- Otherwise, addr to addr+length bytes are synchronized.
- msync() accepts the following flags:

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.
- length=0: all pages containing addr are synchronized.
- Otherwise, addr to addr+length bytes are synchronized.
- msync() accepts the following flags:

  MS_SYNC flush data from mapped region to hard disk; wait until finished

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.
- length=0: all pages containing addr are synchronized.
- Otherwise, addr to addr+length bytes are synchronized.
- msync() accepts the following flags:

  MS_SYNC flush data from mapped region to hard disk; wait until finished

  MS_ASYNC flush data from mapped region to hard disk; don't wait until finished

Memory Mapped I/O: msync()

```
#include <sys/mman.h>
int msync(void *addr, size_t length, int flags);
```

- All modifications, buffers, etc. are moved at the end of this call to an actual physical storage medium.
- length=0: all pages containing addr are synchronized.
- Otherwise, addr to addr+length bytes are synchronized.
- msync() accepts the following flags:

  MS_SYNC flush data from mapped region to hard disk; wait until finished

  MS_ASYNC flush data from mapped region to hard disk; don't wait until finished

  MS_INVALIDATE invalidate data in mapped region; subsequent access will cause new pages from the hard disk to be used.

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from `addr` to `addr+len`

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from `addr` to `addr+len`

```
int mlockall(int flags);
int munlockall(void);
```

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from addr to addr+len

```
int mlockall(int flags);
int munlockall(void);
```

- mlockall() Locks all pages mapped into the address space of the calling process. These pages are locked into RAM; they are not swapped out. (this includes code, data, stack, shared libraries, user space kernel data, shared memory, and memory-mapped files)

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from addr to addr+len

```
int mlockall(int flags);
int munlockall(void);
```

- mlockall() Locks all pages mapped into the address space of the calling process. These pages are locked into RAM; they are not swapped out. (this includes code, data, stack, shared libraries, user space kernel data, shared memory, and memory-mapped files)
  - Flags:

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from addr to addr+len

```
int mlockall(int flags);
int munlockall(void);
```

- mlockall() Locks all pages mapped into the address space of the calling process. These pages are locked into RAM; they are not swapped out. (this includes code, data, stack, shared libraries, user space kernel data, shared memory, and memory-mapped files)
  - Flags:

    MCL_CURRENT lock all pages that are currently mapped

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from addr to addr+len

```
int mlockall(int flags);
int munlockall(void);
```

- mlockall() Locks all pages mapped into the address space of the calling process. These pages are locked into RAM; they are not swapped out. (this includes code, data, stack, shared libraries, user space kernel data, shared memory, and memory-mapped files)
  - Flags:

    MCL_CURRENT lock all pages that are currently mapped

    MCL_FUTURE lock all pages which become mapped in the future

Memory Mapped I/O: mlock() and munlock()

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

- Locks/unlocks part or all of the calling process' virtual address space into RAM (preventing its being written out to swap memory or hard disk)
- Lock/unlocks pages from addr to addr+len

```
int mlockall(int flags);
int munlockall(void);
```

- mlockall() Locks all pages mapped into the address space of the calling process. These pages are locked into RAM; they are not swapped out. (this includes code, data, stack, shared libraries, user space kernel data, shared memory, and memory-mapped files)
  - Flags:

    MCL_CURRENT lock all pages that are currently mapped

    MCL_FUTURE lock all pages which become mapped in the future
- munlockall() unlocks all pages mapped into the address space of the calling process.

IPC - Shell Interface

- ipcs [-q|-m|-s]

  Provides ipc status for (q) message queues

IPC - Shell Interface

- ipcs [-q|-m|-s]

  Provides ipc status for (q) message queues

  (m) shared memory

IPC - Shell Interface

- ipcs [-q|-m|-s]

  Provides ipc status for (q) message queues

  (m) shared memory

  (s) semaphores

IPC - Shell Interface

- ipcs [-q|-m|-s]

  Provides ipc status for (q) message queues

  (m) shared memory

  (s) semaphores

  No options; this command will print information for all three types of IPC.

IPC - Shell Interface

- ipcs [-q|-m|-s]

  Provides ipc status for (q) message queues

  (m) shared memory

  (s) semaphores

  No options; this command will print information for all three types of IPC.

- ipcrm [-q msqid|-m shmid|-s semid]

IPC - Shell Interface

- ipcs [-q|-m|-s]

  Provides ipc status for (q) message queues

  (m) shared memory

  (s) semaphores

  No options; this command will print information for all three types of IPC.

- ipcrm [-q msqid|-m shmid|-s semid]
- ipcrm [-Q msgkey|-M shmkey|-S semkey]

  Removes zero or more message queues, semaphore sets, or shared memory segments.